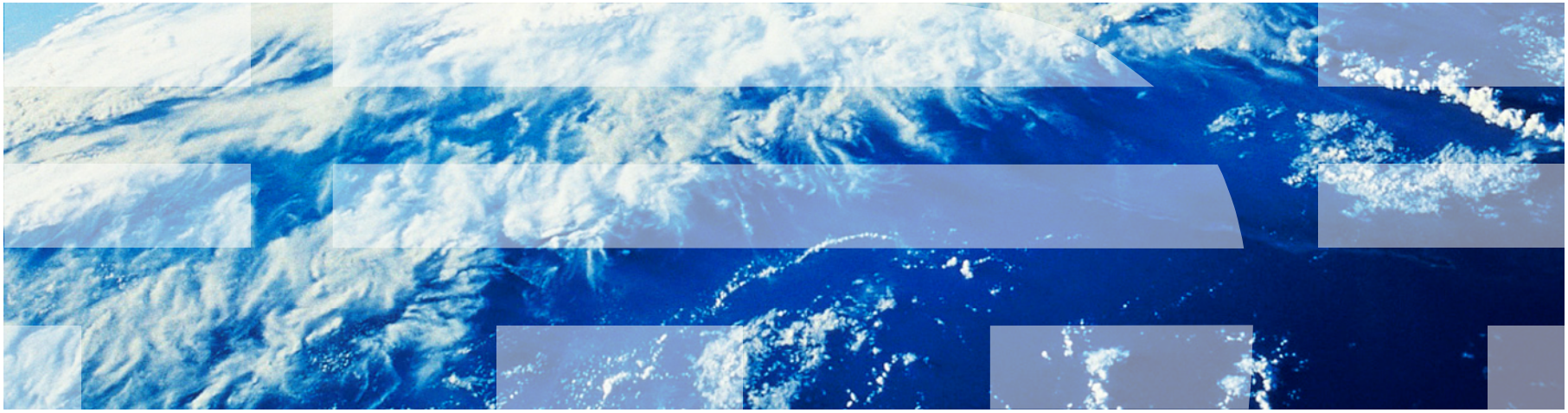
Computer Systems for Data Science
Topic 7

Caching



Motivation

- All the systems we discussed use caching
  - Databases cache frequently accessed keys in memory
  - Spark RDDs are a cache
  - Computers use memory as a cache
  - The file system has a cache, called a page cache
  - …

- Big data systems even use dedicated cache servers (key-value caches)
  - All data is stored in memory (does not use disk)
  - Before going to database, check key-value cache
  - Examples:
    - Memcached
    - Redis
    - Amazon Elasticache

Important questions for caching

- Assignment: where do you put the data?
  - In memory?
    - Where?
  - On disk?
    - Where?
- Consistency: what happens when data is updated/deleted?
- **Eviction: who do you kick out?**
  - **In other words: what do you do when memory is full?**

Standard Eviction Policy Model

- Cache has fixed size (C)
  - Each item has the same size (1)
  - Each item has a unique key
- Cache is full
- Requests come in
  - If the request is for an item that exists in the cache, it is read
  - If the request is for an item that does not exist in the cache, it is fetched (from a second storage system, e.g., a database), and another item is evicted to make room for it
- Eviction policy decides which object to evict when a new item is inserted

Metric for success

- Overall goal might be total latency or throughput (usually latency)
  - E.g., what's the performance of the total system given a certain eviction policy
- Typically average latency is equivalent to cache hit rate
  - Over time window, percentage of read requests whose objects existed in the cache
- Example (includes the network latency):
  - Average latency to flash database: 5 millisecond
  - Average latency to in-memory key-value cache: 100 microseconds
  - Average latency (in microseconds) when cache hit rate is 98%:
    - 0.98 * 100 + 0.02 * 5000 = 198 microseconds
  - Average latency (in microseconds) when cache hit rate is 99%:
    - 0.99 * 100 + 0.01 * 5000 = 149 microseconds
  - 1% cache hit rate increase improves latency by 33%!

Common eviction policies

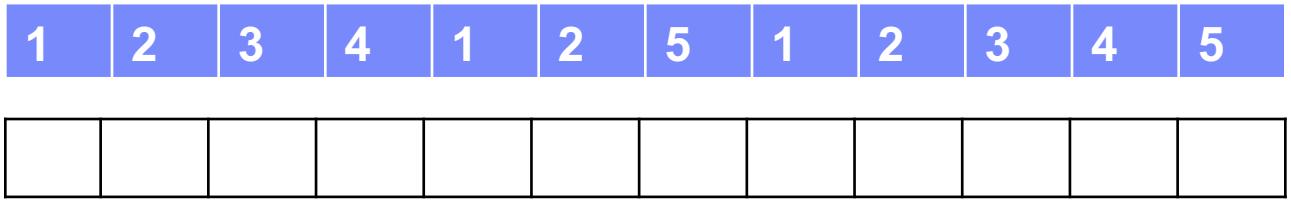
- FIFO (first in first out)
- LRU (least recently used)
- CLOCK (approximates LRU)
- LFU (least frequently used)
- OPT (Belady's algorithm: knows the future)

FIFO (First In First Out)

- Evict the item that was **last inserted** into the cache
- Idea: the item that was last inserted is the oldest, and hence the least relevant
- Advantages:
  - Simple and easy to implement
  - Can be implemented as a list, where new item are inserted to the tail of the list, and items are evicted from the head
- Why might it work?
  - Sometimes the item that was brought in the cache last is old and is not read anymore
- Why might it not work?
  - Because maybe an old item is actually a popular item that is accessed frequently
- In general, usually suffers from a low hit rate

FIFO example

- List of cache accesses (by key):

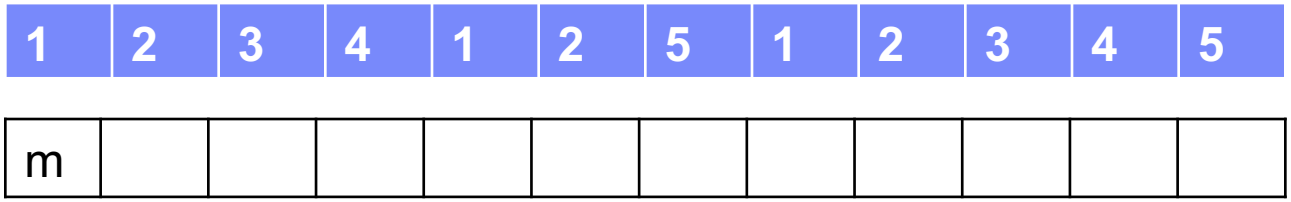


- Cache (size 4):

- (leftmost item is the next to be evicted)

FIFO example

- List of cache accesses (by key):

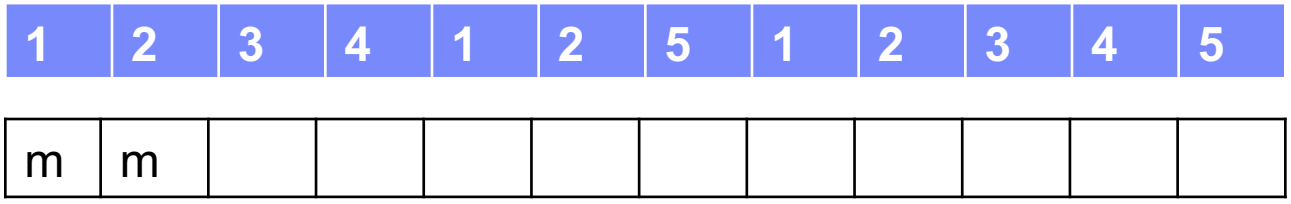


- Cache (size 4):

- (leftmost item is the next to be evicted)

FIFO example

- List of cache accesses (by key):



- Cache (size 4):

- (leftmost item is the next to be evicted)

FIFO example

- List of cache accesses (by key):

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m								

- Cache (size 4):

1	2	3	4

FIFO example

- List of cache accesses (by key):

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h						

- Cache (size 4):

1	2	3	4

FIFO example

- List of cache accesses (by key):

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m					

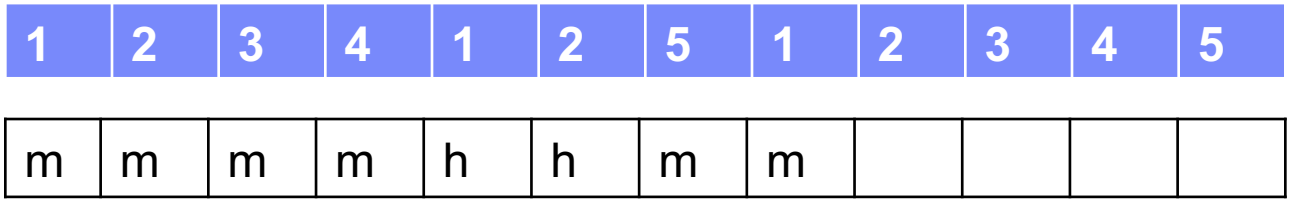
1 evicted

- Cache (size 4):

2	3	4	5

FIFO example

- List of cache accesses (by key):



2 evicted

- Cache (size 4):

FIFO example

- List of cache accesses (by key):

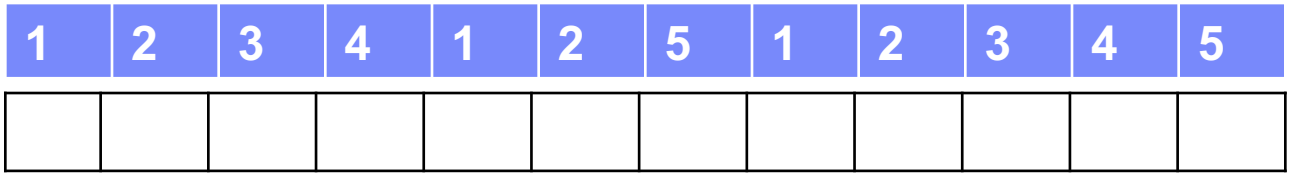
1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	m	m	m	m	m

Total # misses: 10

LRU (Least Recently Used)

- Evict the object that was **last accessed** from the cache
- Idea: recently accessed item likely to be accessed again in the future
- How is LRU different than FIFO?
- Advantages:
  - The default eviction policy, works reasonably well for many workloads
- Disadvantages:
  - Sounds simple, but can be expensive to implement
  - Need to track the last access time of each item, and find item with the oldest access item

LRU example



LRU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m								

1	2	3	4

LRU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h							

2	3	4	1

Need to rearrange cache order even if item is hit

LRU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h						

3	4	1	2

LRU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m					

3 is evicted

4	1	2	5

LRU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h				

4	2	5	1

LRU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h			

4	5	1	2

LRU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h	m		

4 is evicted

5	1	2	3

LRU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h	m	m	

5 is evicted

1	2	3	4

LRU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h	m	m	m

1 is evicted

2	3	4	5

Total: 8 misses
Better than FIFO, but not optimal…

LRU Approximations

- LRU is expensive to fully implement
  - Need to update a list of items at every read
  - Maintaining cache ranking of all items can be expensive
- In practice, many caches use approximations of LRU
- Sampled LRU
  - Instead of maintaining full ranking of all items (in a list), every time a new item is accessed, simply update its access time
  - At eviction time, sample N items, and pick the one with the oldest access time
  - When N is large enough (e.g., 50/100 or more) this approximates LRU very well
- CLOCK
  - Maintain 1 bit per item in the cache (can be 0 or 1)
  - When an item is accessed set bit to 1
  - Occasionally use a clock “hand” that decrements bits to 0 one by one
  - At eviction time, evict next item that has a bit 0
  - This is the algorithm originally used to cache memory pages in the operating system

LFU (Least Frequently Used)

- Evict the object that was **accessed the fewest total time** from the cache
  - Usually # of times over a period time (i.e., frequency)
- Idea: items that were accessed many times likely to be accessed again in the future
- How is LFU different than LRU?
- Advantages:
  - Works well for workloads that have a relatively static number of popular items
- Disadvantages:
  - Does not work well if new items become popular
  - Can be difficult to implement: need to track item frequency for all keys, find the one with the lowest frequency
    - To fully implement, also need to track frequencies of evicted items

LFU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m								

- # accesses:

1	2	3	4
1	1	1	1

LFU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h							

- # accesses:

1	2	3	4
2	1	1	1

LFU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h						

- # accesses:

1	2	3	4
2	2	1	1

LFU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m					

Evict 3/4 (randomly chose 3)

1	2	4	5

- # accesses:

2	2	1	1

LFU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h				

- # accesses:

1	2	4	5
3	2	1	1

LFU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h			

- # accesses:

1	2	4	5
3	3	1	1

LFU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h	m		

Evict 4/5 (randomly chose 5)

- # accesses:

1	2	3	4
3	3	2	1

LFU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h	m	h	

- # accesses:

1	2	3	4
3	3	2	2

LFU example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h	m	h	m

Evict 3/4 (randomly chose 4)

- # accesses:

1	2	3	5
3	3	2	2

Total misses: 7

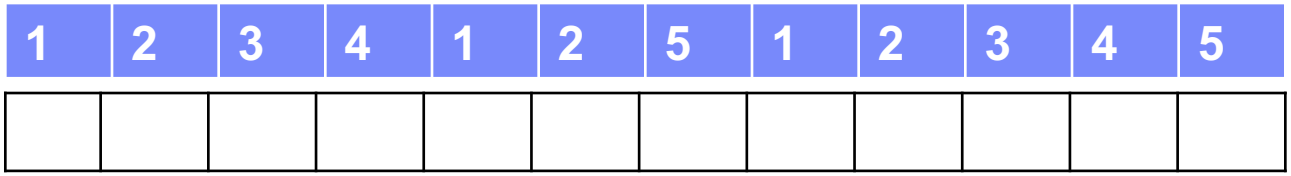
LRU/LFU combinations

- LRU prioritizes recent accesses, while LFU prioritizes frequently accessed items
- Recency vs. frequency are both important features, each works better for different workloads
- Many hybrid frequency and recency algorithms
- For example, Segmented LRU
  - The first time an object is inserted into the cache, insert it into the middle of the cache (not the head)
  - Afterwards, at every access insert it to the top of the cache
  - Mostly behaves like LRU, but penalizes objects that might be accessed recently but just once

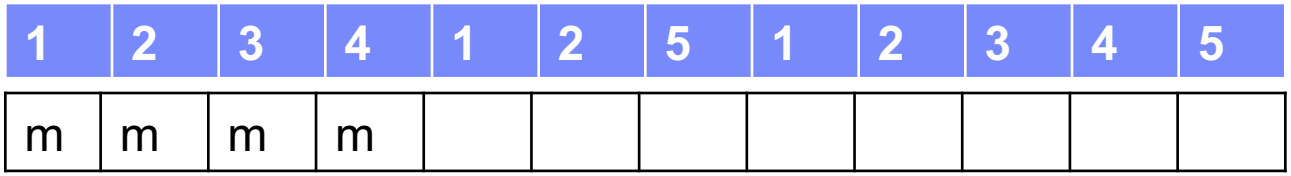
OPT/Optimal (Belady's Algorithm)

- Each one of the algorithms we surveyed earlier can perform better (or worse) on certain workloads
- The Optimal algorithm (invented by Belady) is the optimal eviction algorithm
  - Assumes it knows the future
  - Evicts page that won't be accessed for the longest time in the future
- This is not an algorithm that can be used in practice (since it needs to know the future!)
  - But can be used to evaluate how far off LRU/LFU/etc. are from the optimal for a given workload
  - Some recent work uses OPT to train machine learning eviction policies ("Learning Relaxed Belady", Song et al.)

OPT example



OPT example



OPT example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h							

4	3	1	2

OPT example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h						

4	3	2	1

OPT example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m					

Evict 4 (furthest away in the future)

5	3	2	1

OPT example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h	h		

3	2	1	5

OPT example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h	h	m	

4	2	1	5

OPT example

1	2	3	4	1	2	5	1	2	3	4	5
m	m	m	m	h	h	m	h	h	h	m	h

4	2	1	5

Total misses: 6 (this is the optimal!)

Eviction policies in the real world

- All of the eviction policies we surveyed are used by real systems
  - Except for OPT!
- LRU and its approximations are the most common
- Complications:
  - We assumed all items are the same size
  - Things get more complicated when items can have different sizes
    - E.g., Bigger items might cause the eviction of many small items